csci 3411: Operating Systems

Memory Management II

Gabriel Parmer

Slides adapted from Silberschatz and West

<ubergeek-analogy>

Each Process has its Own Little World

Picture from “The Matrix”, Warner Bros. Pictures

- Virtual Address Space
  - Private memory
  - Process can manage it's own memory
    - Ask kernel for more if needed


If virtual address spaces provide a virtual world,
What is the world, “of the real”
The Matrix, Warner Bros. Pictures

csci3411 == red pill ?

The Matrix, 1999, Warner Bros. Pictures

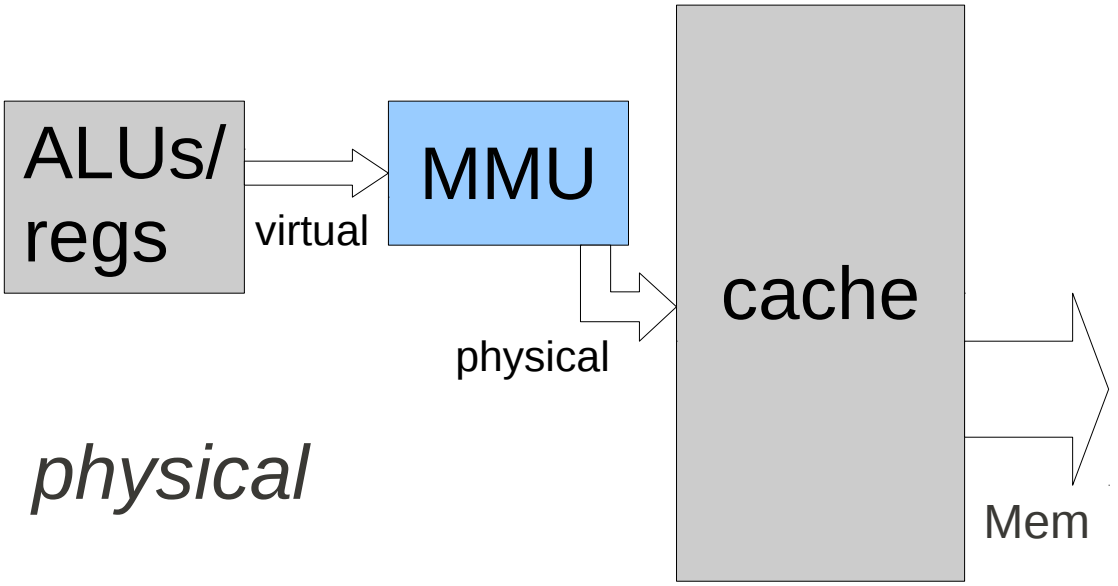
</ubergeek-analogy>

Virtual vs. Physical Address Space

- The memory processes can access is restricted
  - A subset of actual memory
  - Memory a process can access controlled by OS
- Virtual/Logical Address – address of memory generated by the process on CPU
- Physical Address – offset into the physical RAM of memory to access
- *What converts virtual addresses into physical???*

MMU: Memory Management Unit

- Addresses generated by program → virtual
  - MMU: translation to physical
  - Level of indirection



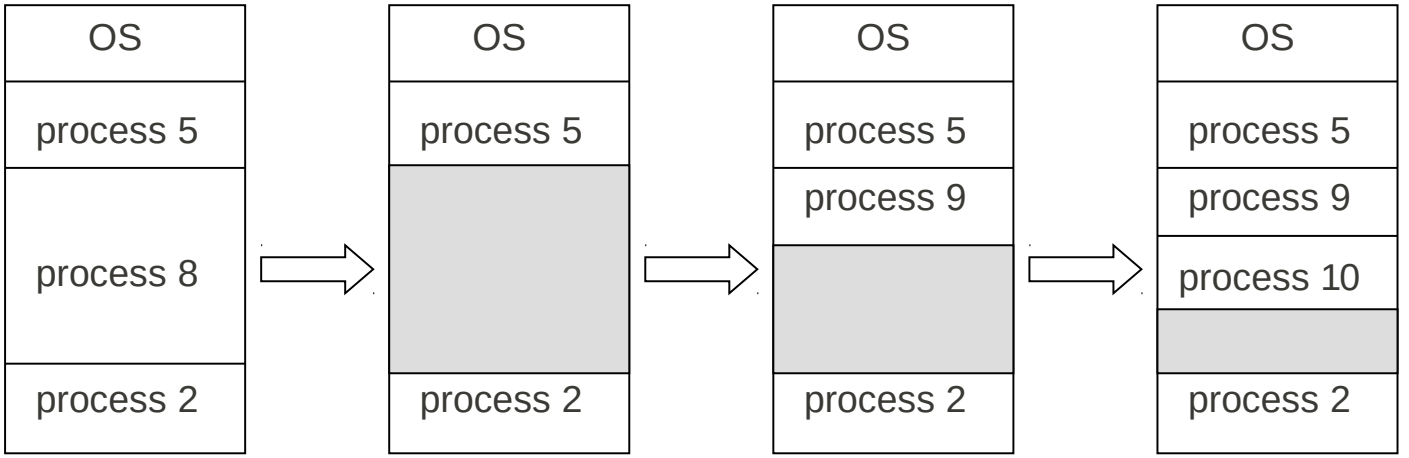
MMU does: *f(virtual)* → *physical*

- *f(virtual, ?)* → *physical*

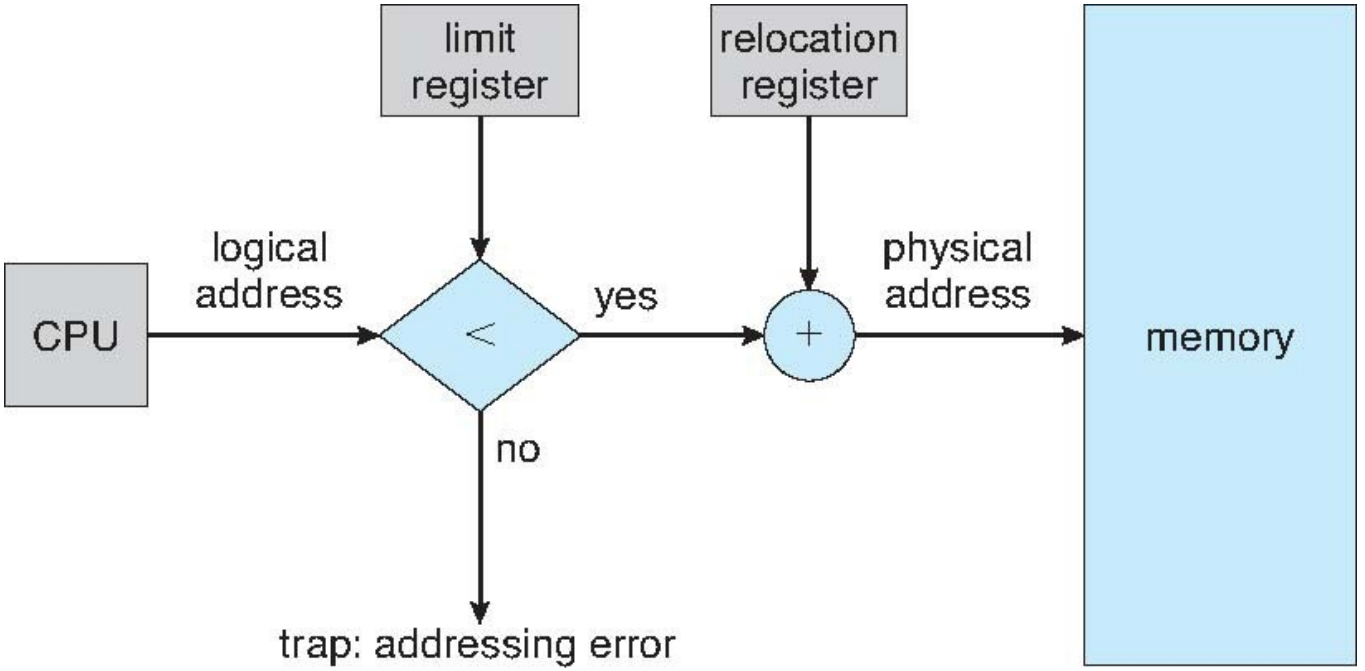
What if the MMU was between cache ↔ *memory?*

Process' Physical Memory

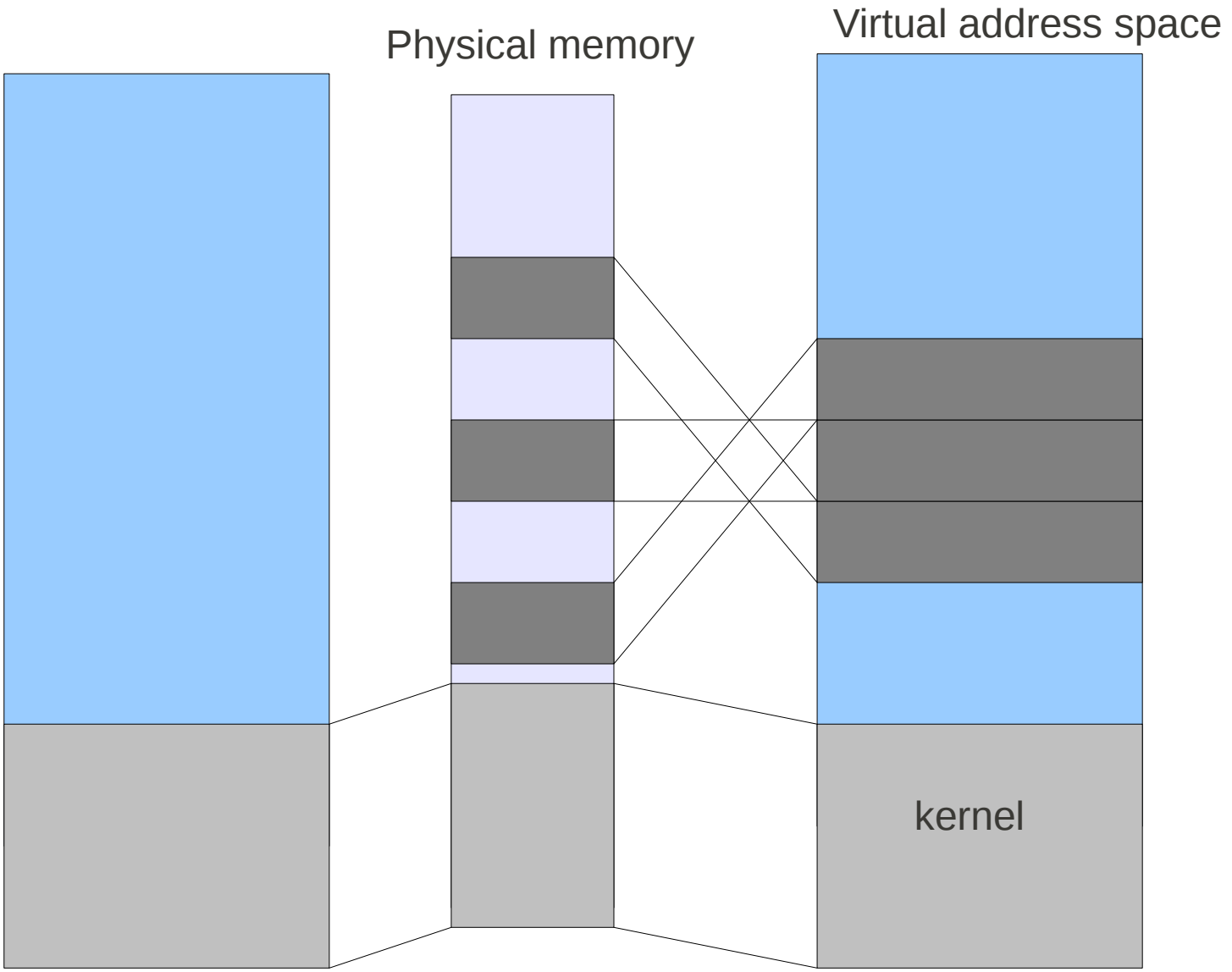
- Contiguous allocation of physical memory to processes?
  - What if a process doesn't use all of its allocation
  - What if it uses more
- Hard to predict exactly how much mem to alloc



MMU Example: Protection + Contiguous Allocation



Contiguous vs. *Non-contiguous*

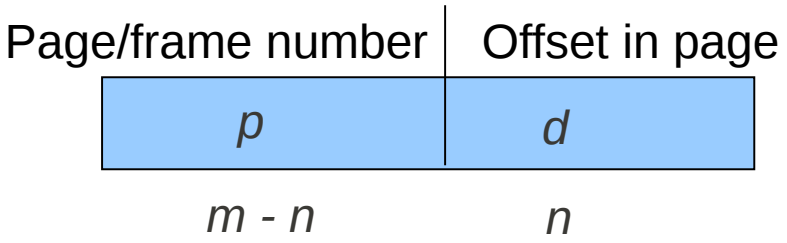


Paging

- Physical memory allocated to process can be non-contiguous
- Divide physical memory into fixed sized *frames*
  - Size is power of 2, x86: two page sizes 4K, 4M
- Virtual memory divided into *pages* (same size)
- Track free *frames*
- When process requests memory, allocate to it a number of frames
  - Internal fragmentation
- MMU translates between *pages* to *frames*

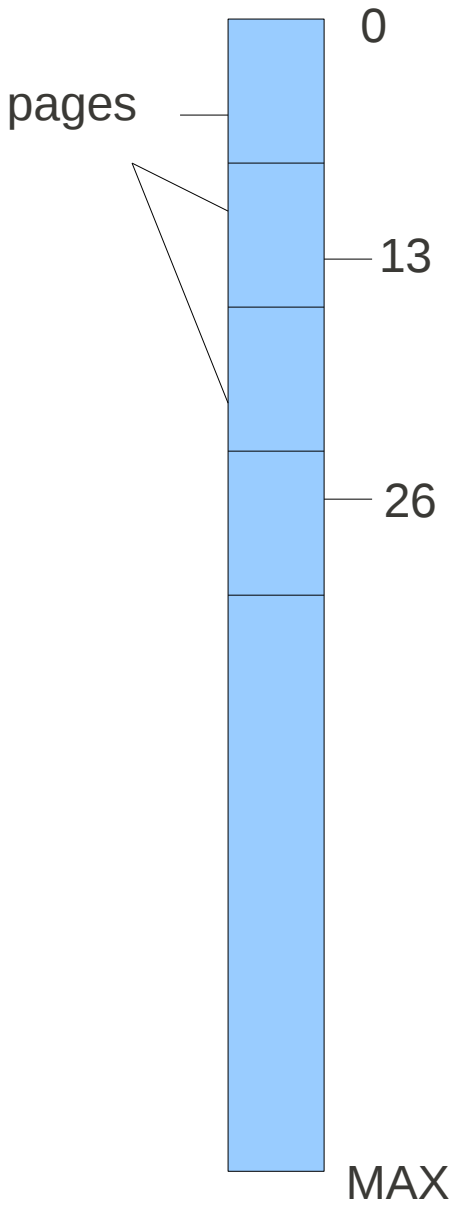
Address Translation Scheme

- Address generated by CPU (virtual) divided into
  - *Page number (p)* – used as an index into a *page table* which contains the base address of each page in physical memory
  - *Page offset (d)* – combined with base address to define the physical memory address that is set to the memory unit
- Logical Address space 2^m, page size 2^n



MMU: $f(p) \rightarrow f + d$

Example: 8B pages

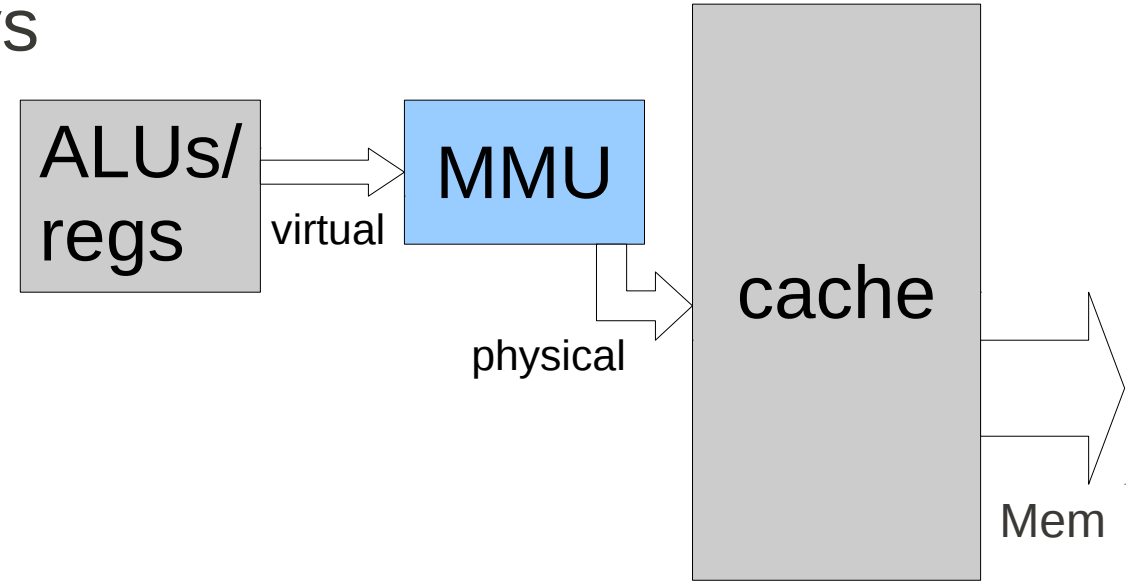


- $2^3 = 8 \rightarrow$ 3 bits represent < 8
- 13 = 1101 (least significant = right)
  - 101 = 5 = offset into page
  - 1 = page number (2^{nd} page)
- 26 = 11010
  - 010 = 2 = offset into page
  - 11 = 3 = page number (4^{th} page)

- *C code for getting page/offset?*

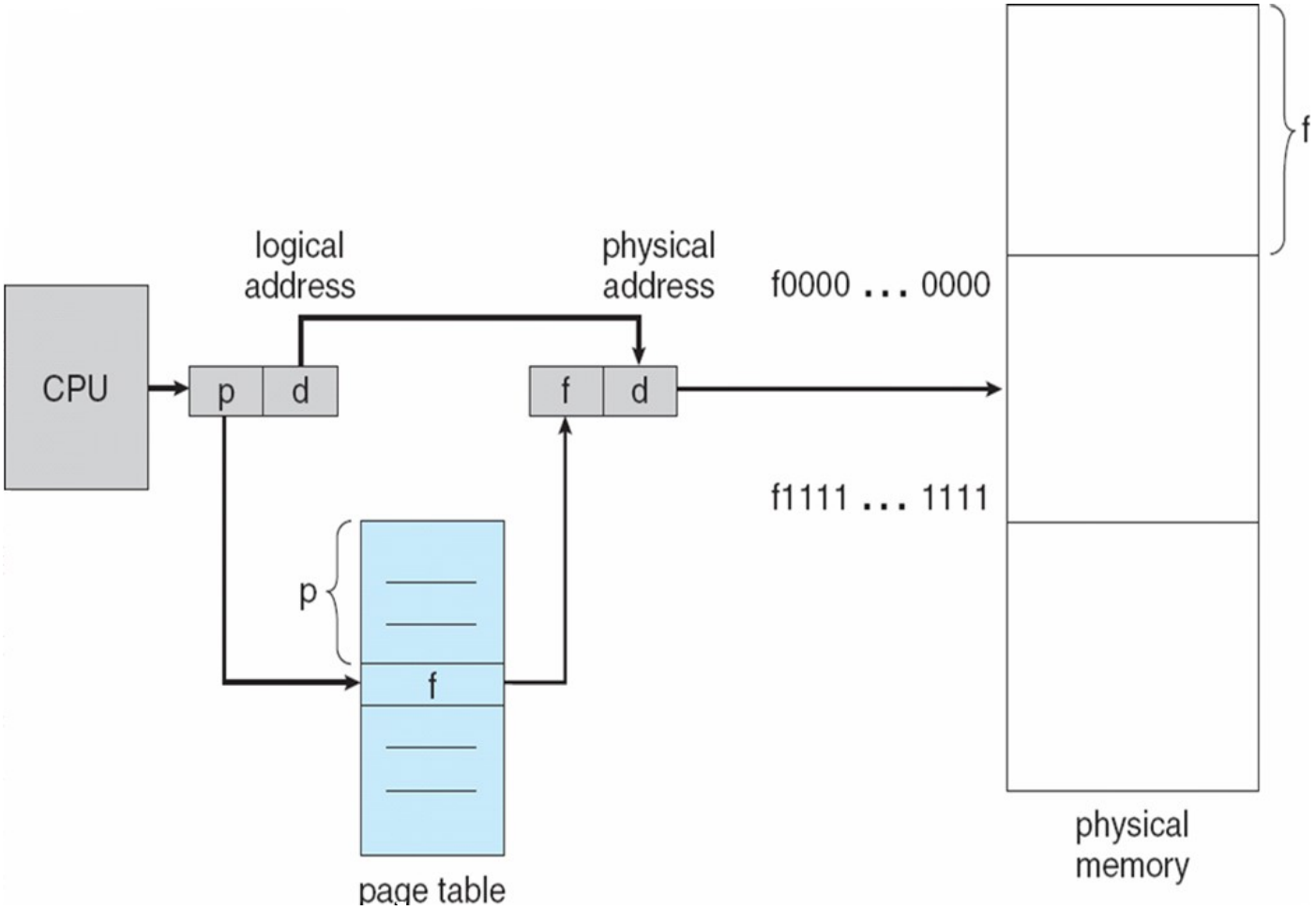
MMU and Paging

- Virtual address split into pages
- Page access translated into physical frames
  - Non-contiguous phys allocation
  - On-demand page allocation
- Level of indirection

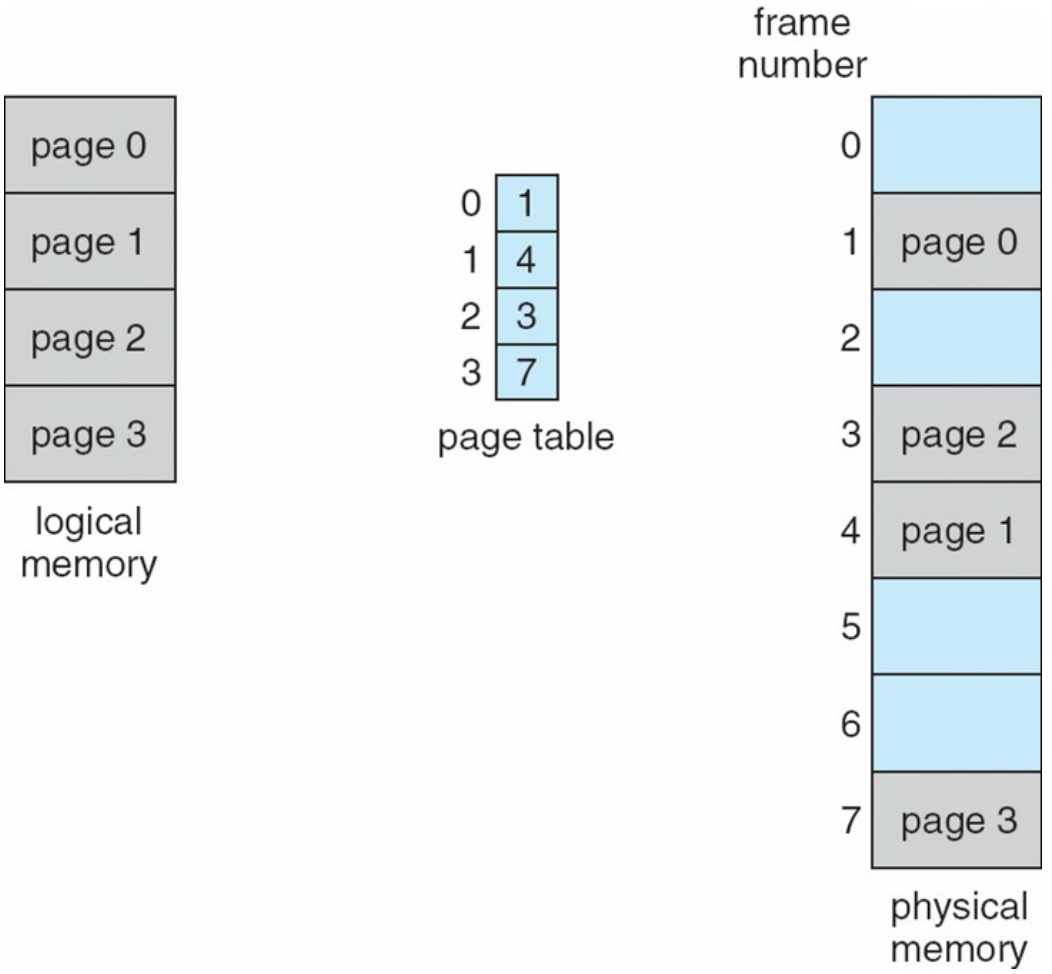


Paging Hardware (MMU)

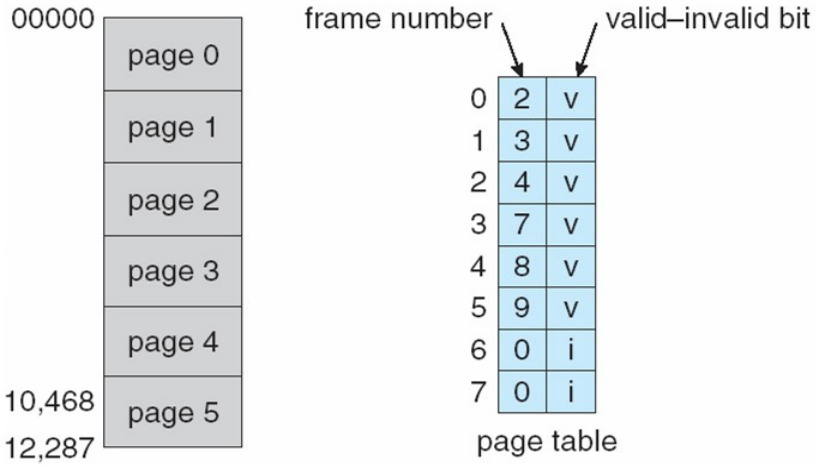
- OS controls page tables

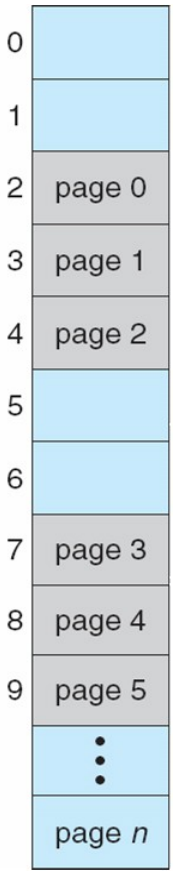


Page Table Translation



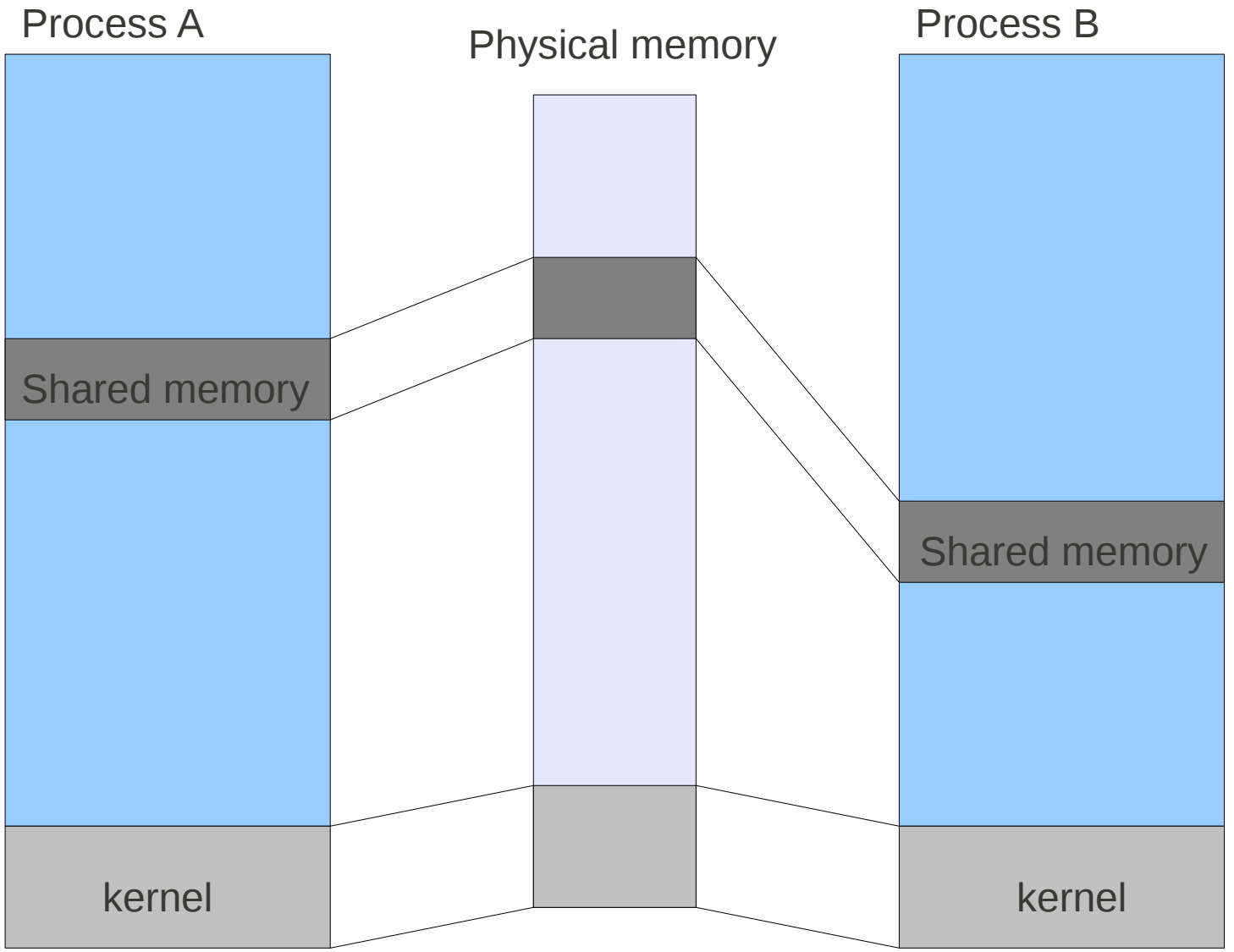
Protection via Page Tables





- What happens if a memory access is made to a virtual address marked as *invalid* or *not present*?
- Other bits in the PT
  - readable
  - writeable (COW!)
  - executable

Shared Memory

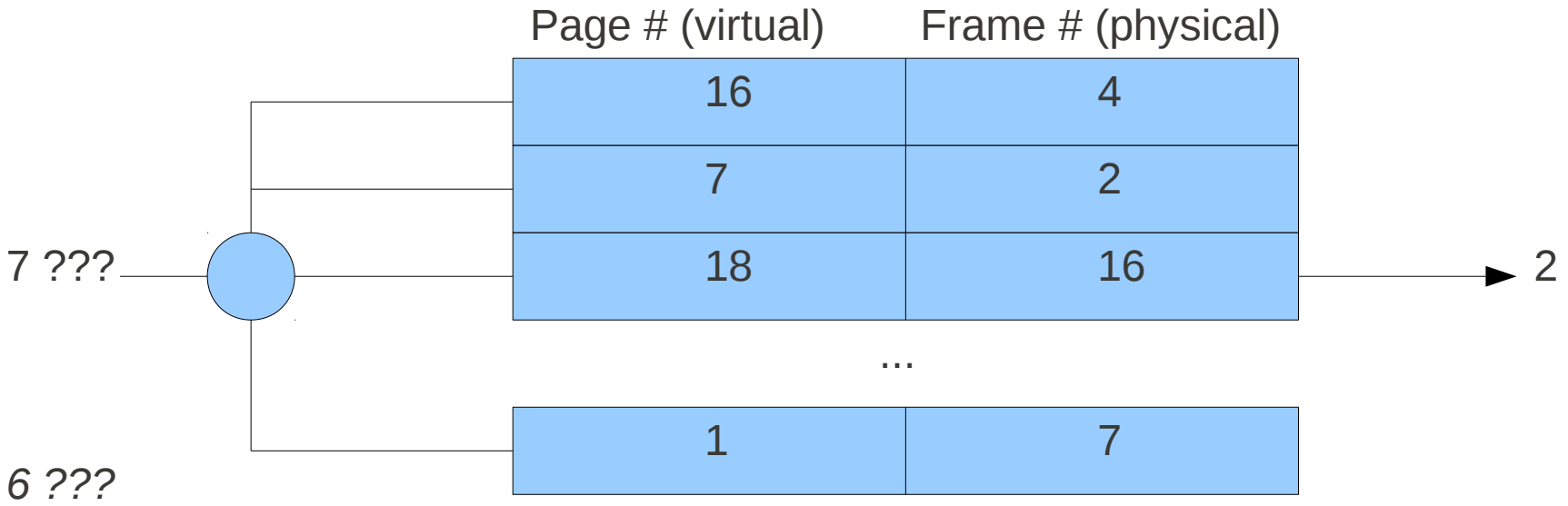


Page Table Implementation

- Page tables kept in main memory
- *Page Table Base Register* (PTBR) contains address of current page table
  - cr3 register on x86
- Privileged instruction to modify the PTBR
  - *Why?*
- *PTBR holds a virtual or physical address?*
- *How many memory accesses for each load/store?*

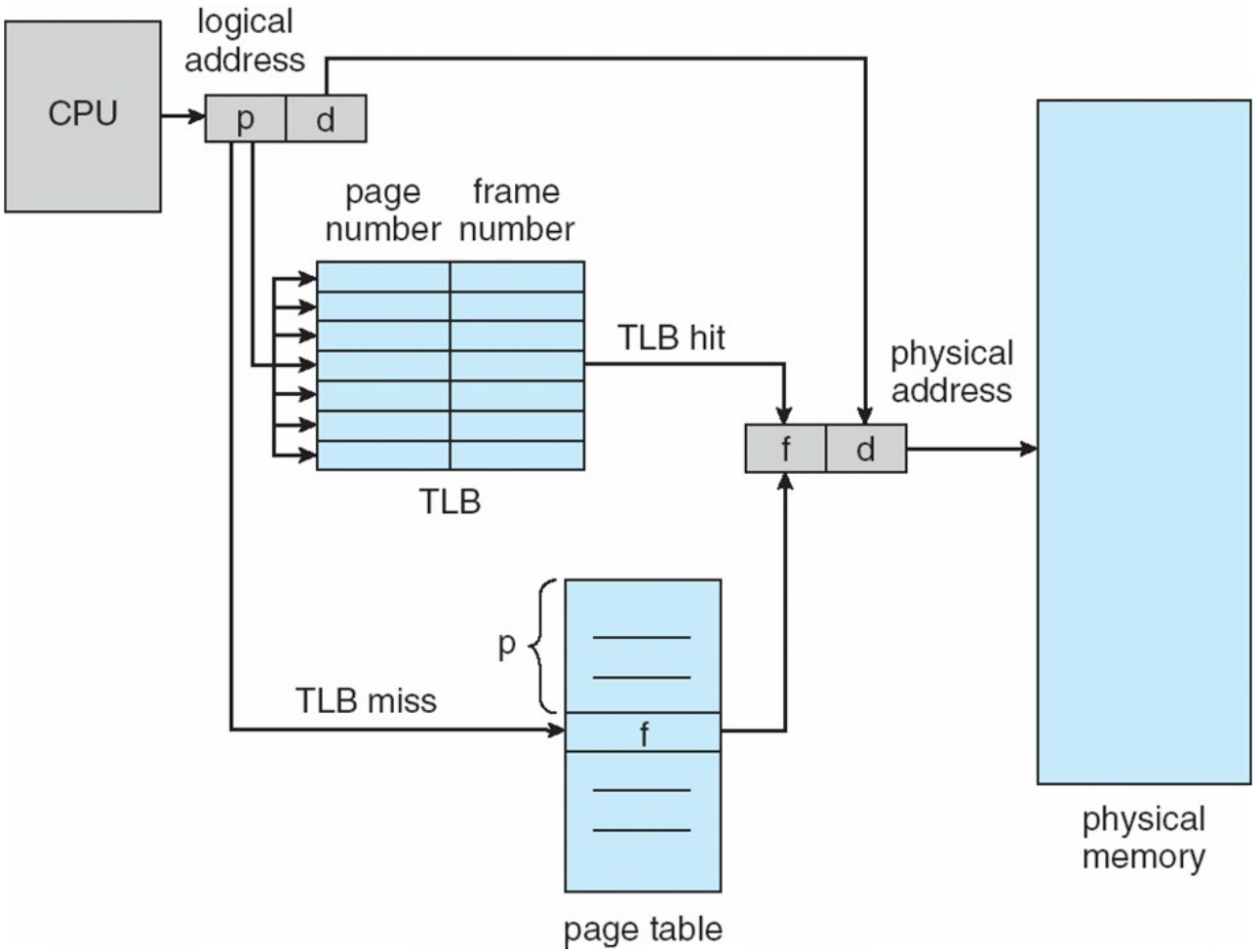
Translation Lookaside Buffer (TLB)

- Cache holding *N* page → frame mappings
  - Associative memory, part of MMU
  - Must provide translation every cycle
  - Reduces number of memory accesses



- *What does the hardware do on a cache miss?*

TLB + Paging


CPU
logical address
p
d
page number
frame number
TLB hit
physical address
f
d
TLB
p
TLB miss
f
page table
physical memory

Memory Access Performance

- M = memory access time
- Without TLB, memory access time = $2M$

- Effective Access Time (EAT):
  - Hit ratio = α, is the % of time page found in TLB
  - TLB search time = δ
  - EAT = $\alpha (M + \delta) + (1-\alpha)(2M + \delta)$

    $= M(2-\alpha) + \delta$

EAT Example

- Hit ratio of 80%
- Memory (cache) access = 100ns
- TLB search = 20ns
- TLB hit = 120ns, miss = 220ns
- EAT = 0.8*120 + 0.2*220 = 140ns
  - 40% slowdown over single memory access
  - Compared to a 100% slowdown for memory access always via page-table

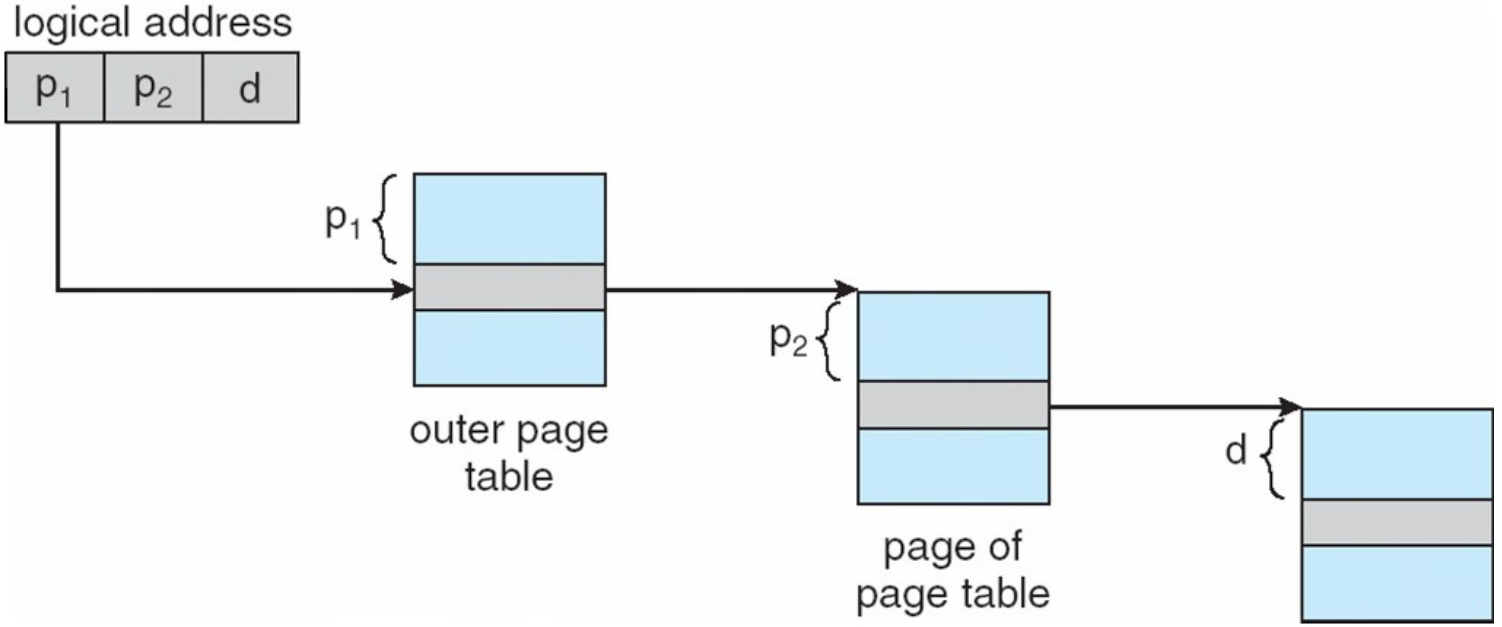
HW vs. SW page-table traversal

- Does hardware contain logic for traversing a specific format of page tables?
  - HW page-table traversal
- SW traversal of page tables:
  - TLB miss → translation fault
    - kernel handler activated
    - Software parses the page tables
    - Tells HW what translation to put in the TLB
- *Tradeoffs?*

Page Table Structures

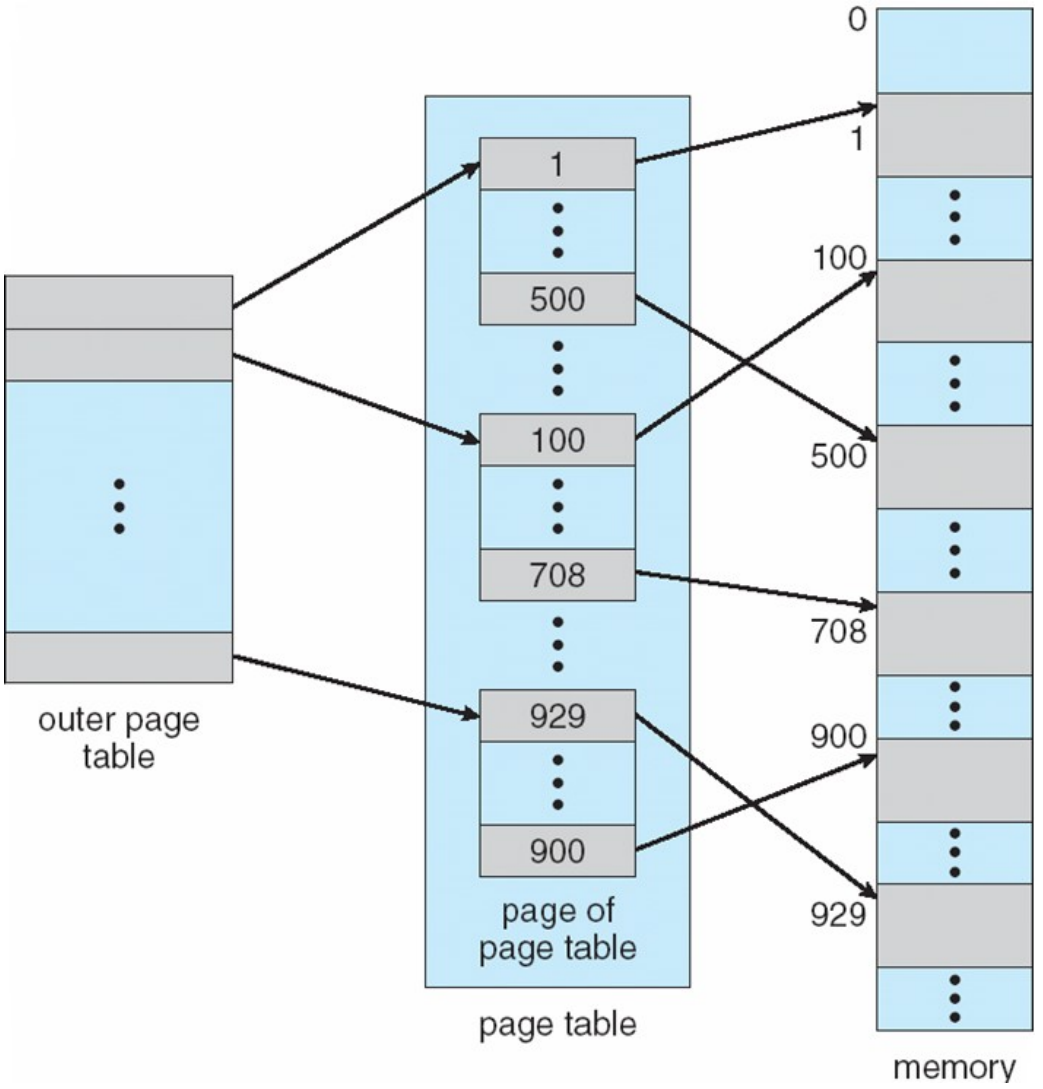
- So far, size of a single page table = # pages in virtual address space
  - 32 bit, 4K pages = 2^{20} pages → page-table entries
    - 4MB memory.....*per-process*
- Typical Practical Structures
  - Hierarchical Page Tables
  - Hashed Page Tables
  - Inverted Page Tables

Two Level Page Tables



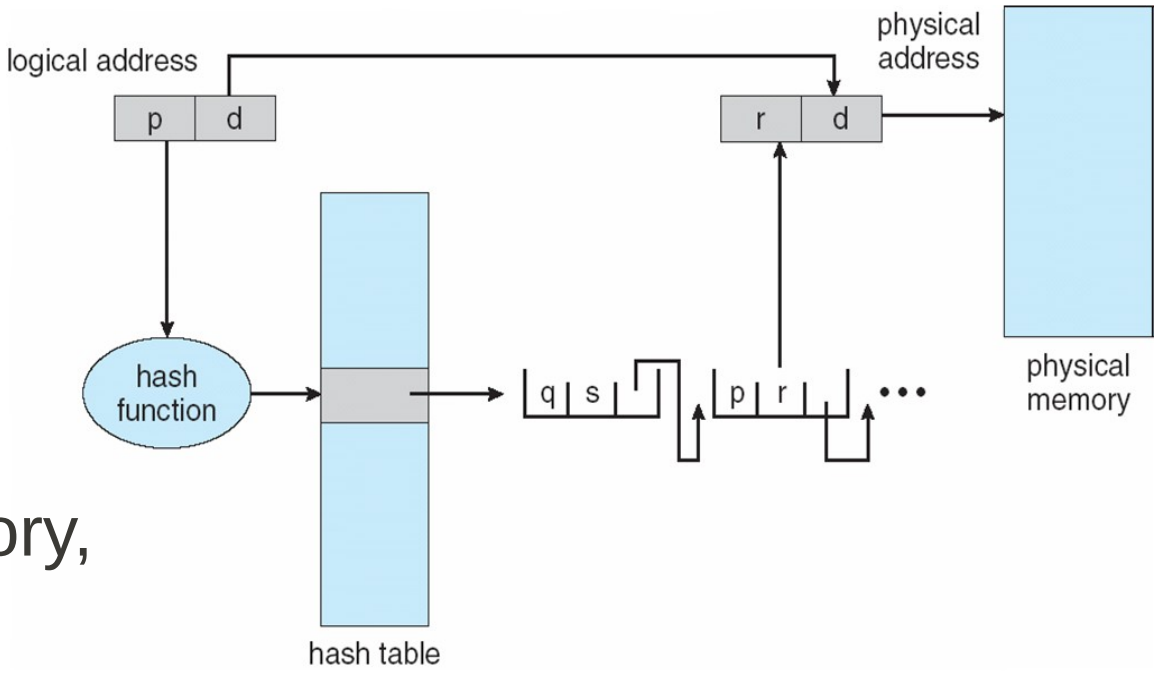
- 32 bit virtual address space, 4K (2^{12}) page size
  - Page offset, d = 2^{12}
  - 2^{20} addressable pages
    - Outer Page Table: p_1 number of entries (often size of page)
    - Second Level of Page Table: p_2 number of entries (often size of page)
    - size(p_1)+ size(p_2) = 20, references 2^{20} addresses
- *Always saves memory? When?*

Two Level, Hierarchical Page Table



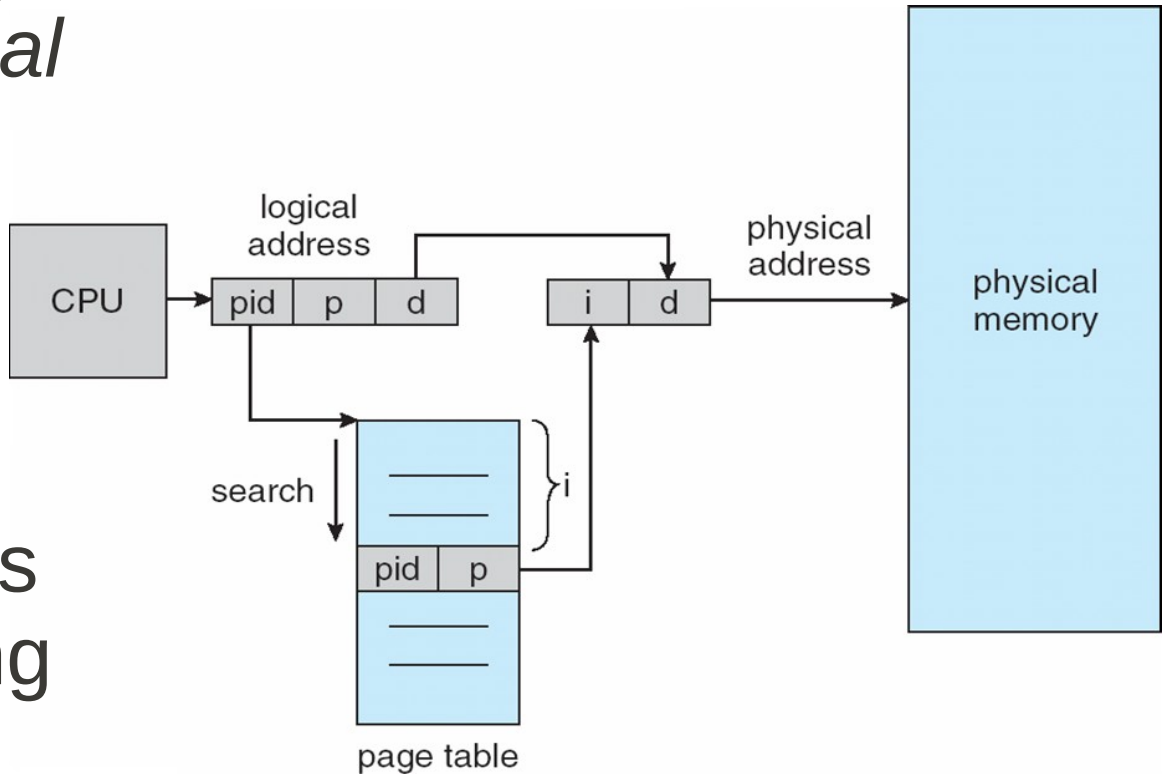
Hashed Page Table

- Hash table stores virtual → physical translations
- Chaining used to resolve conflicts
- Trade-off between size of hash table
  - Large: more memory, faster
- *Worst case overhead?*

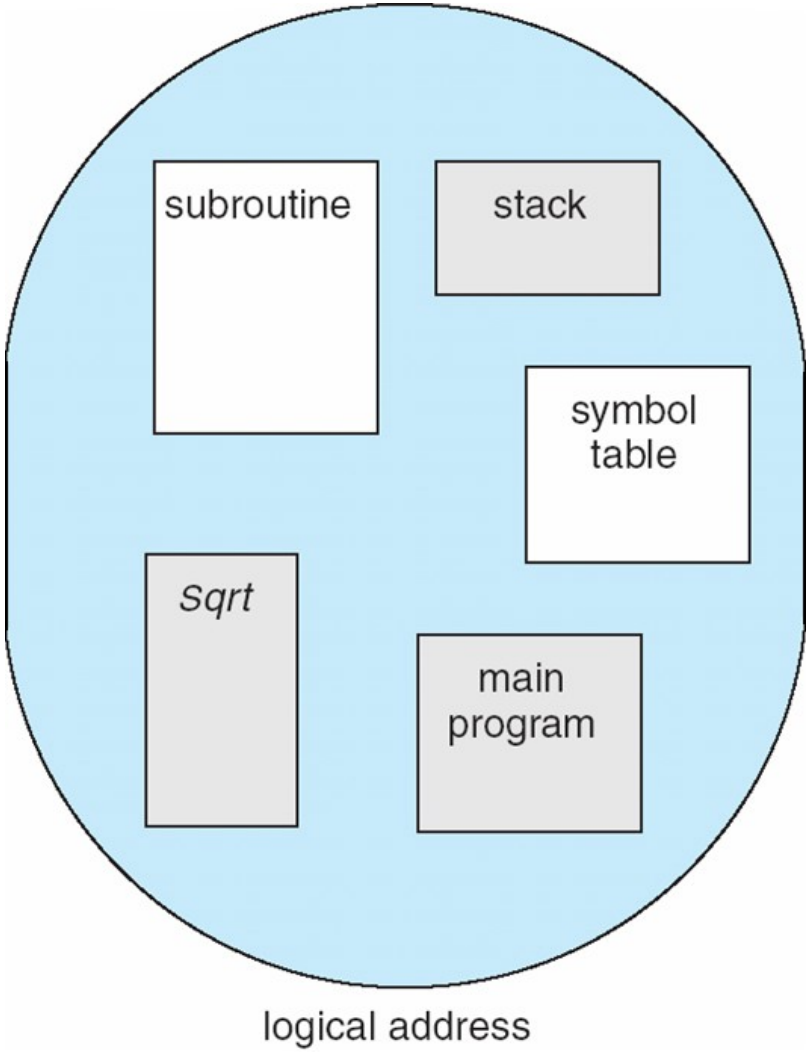


Inverted Page Table

- Page table is array indexed by *physical* frames
  - Entries contain
    - process id
    - *virtual* address
- Linear search finds entry with matching virtual and pid
- One page-table for *all* processes
- *Trade-offs?*



Segmentation

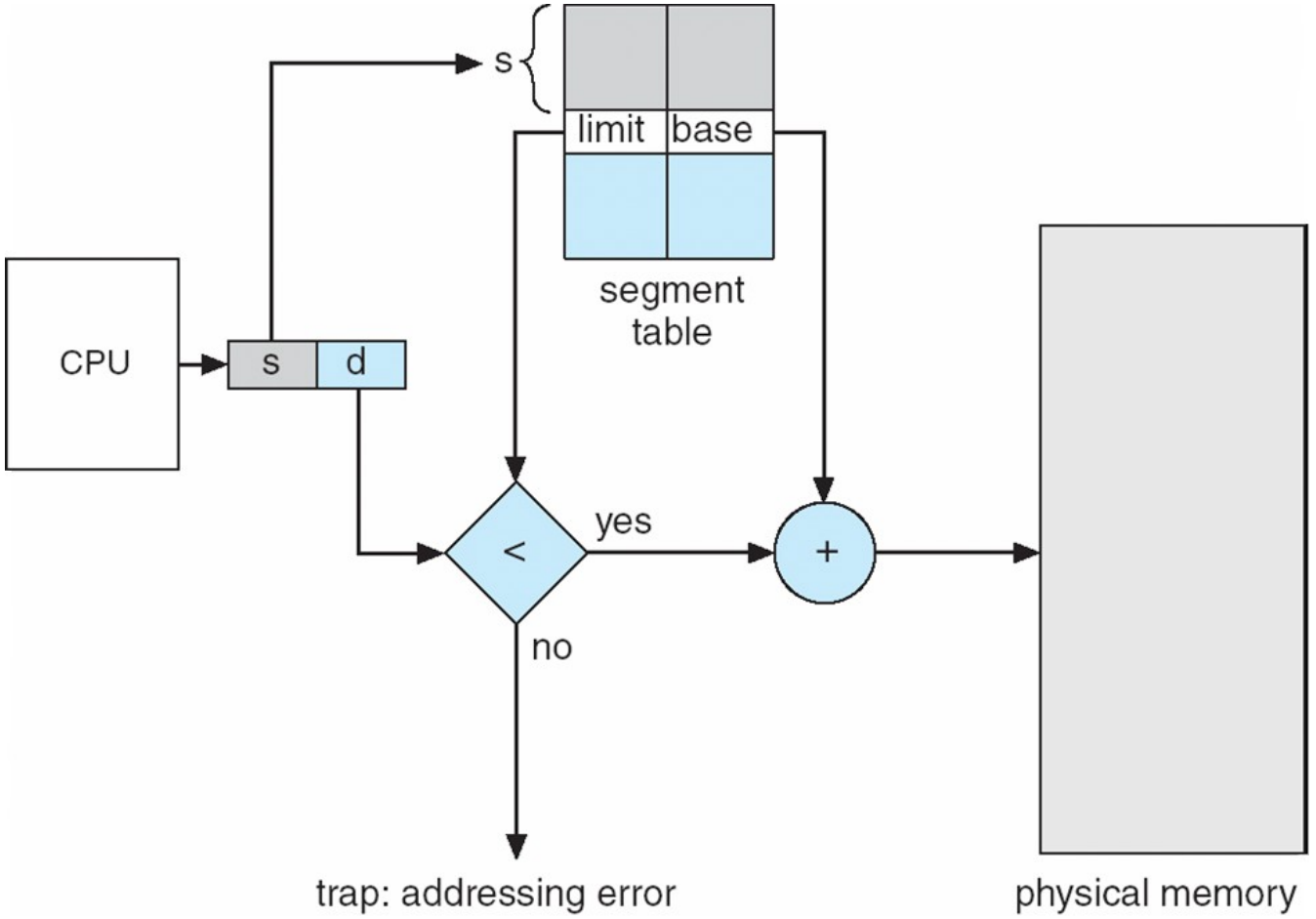


- Program *units* are not page sized!
- Collection of arbitrarily sized *segments*
  - Arrays/data-structures
  - Functions
  - stack

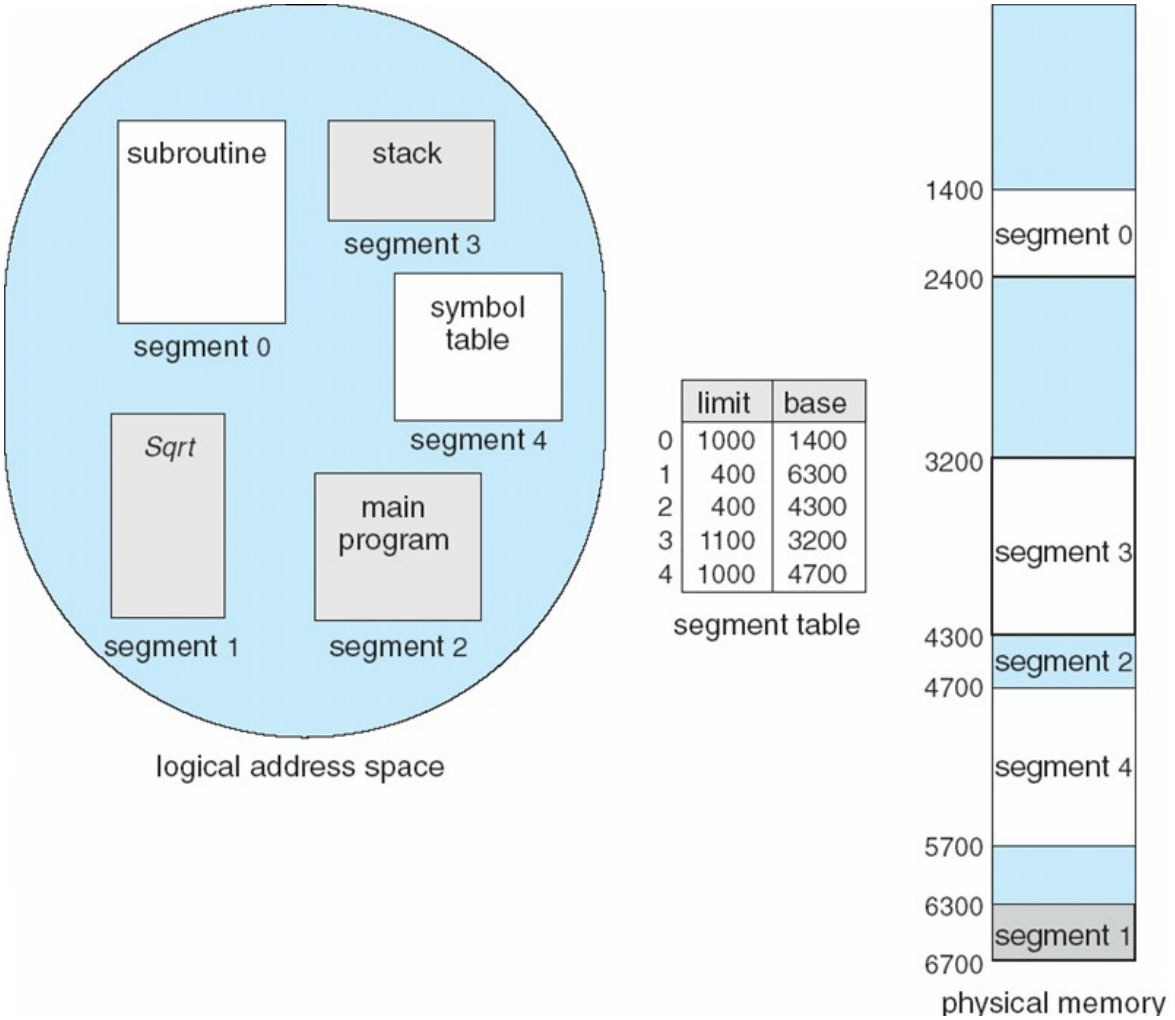
Segmentation II

- virtual/logical address consists of
  - Segment number, and offset
- Segment table translates to physical addresses
  - List of <base, limit> pairs
    - Base: start of segment in physical memory
    - Limit: maximum size of segment
- Segment-table base register

Segmentation IV



Segmentation III



Example Architecture: x86

- Some architectures use both segmentation and paging
  - x86